神經網路梯度下降優化法

簡國龍

May 19, 2021

Contents

Chapter 1

Gradient Descent Optimizer

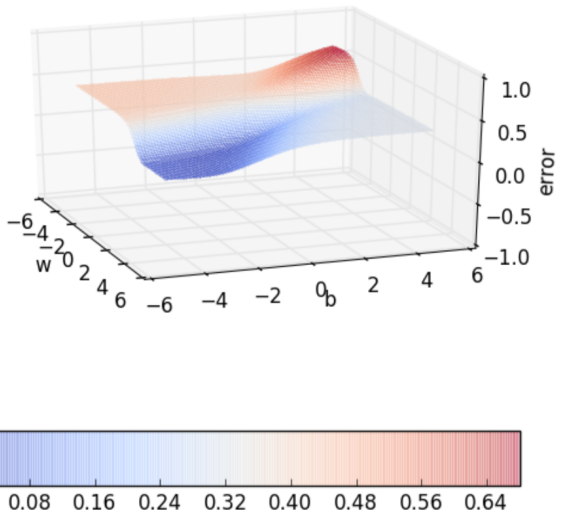

圖. 1.1: Error surface

Gradient descent (梯度下降) 將目標函數 $J_{(\theta)}$(下面是以 loss function $L_{(\theta)}$ 當作目標函數) 最小值化，藉由模型參數 (model's parameters) $\theta \in \mathbb{R}^d$。
假設 θ 是 weight(w) 和 bias(b) 的函數，是一個向量 (matrix)，隨機產生 weight 和 bias 的起始值

$$\theta_{(w,b)} = (w, b)$$

假設 $\overrightarrow{\Delta\theta}$ 是 Δw(權重差值) 和 Δb(偏差差值) 的函數，當 loss 值減少並落在 error 較小的地方，$\Delta\theta$ 就是在降低 loss 值，$\overrightarrow{\Delta\theta}$ 的向量參數就會像 $\Delta\theta \in \mathbb{R}^d$，找到 loss 值最小的地方只需要 θ 朝 $\theta + \Delta\theta$ 方向移動就會找到。

$$\Delta\theta_{(w,b)} = (\Delta w, \Delta b)$$

如果將 $\overrightarrow{\Delta\theta}$ 和 $\overrightarrow{\theta}$ 相加合成成 $\overrightarrow{\theta_R}$(即 [圖.1.2] 上的 $\overrightarrow{\theta_{new}}$)。但要注意 $\overrightarrow{\Delta\theta}$ 的值有可能過大就錯過廣域最小值的地方，所以係數 η 用來縮放 $\overrightarrow{\Delta\theta}$ 的值，η 是純量，通常是小於 1(縮小)。因此

$$\theta_R = \theta + \eta \cdot \Delta\theta$$

θ 剛開始是隨機產生的，但為了確保 $\overrightarrow{\Delta\theta}$ 的方向是朝 loss 值減少的方向，所以要持續循環的計算才能達到廣域 (絕對) 最小值。為了找到正確的 $\overrightarrow{\Delta\theta}$，所以將 θ_R 以泰勒級數方式表現，設 $\overrightarrow{\Delta\theta} = u$

$$L_{(\theta+\eta u)} = L_{(\theta)} + \eta u^T \cdot \bigtriangledown_\theta L_{(\theta)} + \frac{\eta^2}{2!} u^T \cdot \bigtriangledown^2 L_{(\theta)} u + \frac{\eta^3}{3!} ... + \frac{\eta^4}{4!} ... + \frac{\eta^n}{n!} ...$$

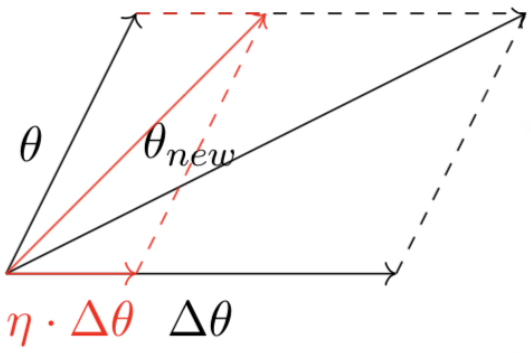

圖. 1.2: theta vector

從泰勒級數中得知 θ 在特定值並做很小的更動，loss function($L_{(\theta)}$) 就會產生新的值，大幅減少每次更動 θ 所計算廣域最小值的次數 (若不是以泰勒級數的型式描述，當每次更動 θ 就必需跌代出廣域最小值；泰勒級數的型式則會隨 θ 些微的更動產生新值)

$$\bigtriangledown L_{(\theta)} = [\frac{\partial L_{(\theta)}}{\partial w}, \frac{\partial L_{(\theta)}}{\partial b}]$$

η 值通常小於一，當 $\eta^2 << 1$，因此可以忽略高階項 (若要求更高的精度可不忽略)

$$L_{(\theta+\eta u)} = L_{(\theta)} + \eta u^T \cdot \bigtriangledown_\theta L_{(\theta)} [\eta \ is \ typically \ small, so \ \eta^2, \eta^3, \cdots \to 0]$$

新的 $L(\theta + \eta u)$ 輸出的值會小於 $L_{(\theta)}$

$$L_{(\theta+\eta u)} - L_{(\theta)} < 0$$

同理可證

$$u^T \cdot \bigtriangledown_\theta L_{(\theta)} < 0$$

符合這條件 u : 當新的值小於舊的值，u 就是一個好的值。假設 $\vec{u}$ 和 $\overrightarrow{\bigtriangledown_\theta L_{(\theta)}}$ 的夾角為 β

$$\cos(\beta) = \frac{u^T \cdot \bigtriangledown_\theta L_{(\theta)}}{|u^T||\bigtriangledown_\theta L_{(\theta)}|}$$

因為 $\cos_{(\theta)}$ 的值介於 1 和-1 之間

$$-1 < \cos(\beta) = \frac{u^T \cdot \bigtriangledown_\theta L_{(\theta)}}{|u^T||\bigtriangledown_\theta L_{(\theta)}|} \leq 1$$

$$assume \ k = |u^T||\bigtriangledown_\theta L_{(\theta)}| \ the \ inequalities \ simplify \ to$$

$$-k \leq k\cos(\beta) = u^T \cdot \bigtriangledown_\theta L_{(\theta)} \leq k$$

所以盡可能的讓新值小於舊值 ($L_{(\theta+\eta u)} - L_{(\theta)} < 0$)，loss 值就會減少得越多。因此 $u^T \cdot \bigtriangledown_\theta L_{(\theta)}$ 應該盡可能為負，在這情況下 $\cos(\beta)$ 應該等於 -1，β 的角度為 180°，這就是與梯度方向相反的原因。

梯度下降法告訴我們：當 θ 在特定值，並想減少新的 θ 值，使 loss 值逐漸減少就應該與梯度相反的方向找 (梯度為正值，找最小值就需往負的方向找)

$$w_{t=1} = w_t - \eta \bigtriangledown w_t$$

$$b_{t=1} = b_t - \eta \bigtriangledown b_t$$

$$where \ at \ w = w_t, b = b_t$$

$$\begin{cases} \bigtriangledown w_t = \frac{\partial L_{(\theta)}}{\partial w} \\ \bigtriangledown b_t = \frac{\partial L_{(\theta)}}{\partial b} \end{cases}$$

Chapter 2

Stochastic gradient descent

2.1 Batch gradient descent

Vanilla gradient descent 又稱 Batch gradient descent(批次梯度下降法)，計算成本函數的梯度，參數 θ 對於整個訓練資料：

$$\theta = \theta - \eta \cdot \bigtriangledown_\theta L_{(\theta)}$$

成本函數可以當作 loss function，因此直接以 $L_{(\theta)}$ 表示，參數 θ 為 weight 和 bias 的函數，η 為學習率。由於計算整個資料集計算梯度只更新一次，Bath gradient descent 可能非常慢並且對於資料集無法符合及記憶體來說棘手 (一次需要儲存整個資料集的資料，當更新和計算時會占用大量記憶體)。他也無法使用在在線更新模型，就是無法即時有新的範例。Bath gradient descent 寫成程式碼 [程式.2.1] 就會像這樣：

程式. 2.1: Batch gradient descent

```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

預定義每次 epoch，先計算 loss function 梯度向量對於整個資料集參數向量。最先進的深度學習資料庫備自動區別功能，對於一些參數有效地計算梯度。如果梯度值來自於先前計算出的梯度值，那檢查就會梯度，並以梯度相反的方向更新參數，學習率決定多大的更新量。Batch gradient descent 對於凸面誤差可以保證收斂到廣域最小值，對於非面凸誤差可以收斂到局部最小值。

2.2 Stochastic gradient descent

Stochastic gradient descent(SGD) 隨機梯度下降法，這裡的目標函數為 $J_{(\theta,x^i,y^i)}$(變數 θ 為 w(weight) 和 b(bias) 的函數，也可以寫成 $J_{(w,b,x^i,y^i)}$)。

$$\theta = \theta - \eta \cdot \bigtriangledown_\theta J_{(\theta,x^i,y^i)}$$

批量梯度下降他會在每個參數更新前重新計算相似梯度。SGD 每次次執行會更新來消除多餘 (誤差)，因此通常速度很快，也可用於在線學習。SGD 頻繁更新並變化很大，因為目標方程式波動很大 [圖.2.1]。SGD 的方程式一方面會跳到新的值和潛在局部最小值，另一方面 SGD 會持續超調 (誤差超過預期) 最後收斂到廣域最小值。

無論如何他被顯示當學習率下降緩慢，SGD 顯示與 Batch gradient descent 同樣收斂行為，幾乎可以肯定地，對於凸面或非凸面優化，會收斂到絕對或是局部最小值。

這程式碼片段 [程式.2.2] 在訓練樣本上加入一個迴圈來對每個樣本評估梯度。每個 epoch(訓練循環) 會打亂訓練數據。

程式. 2.2: Stochastic gradient descent

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params - learning_rate * params_grad
```

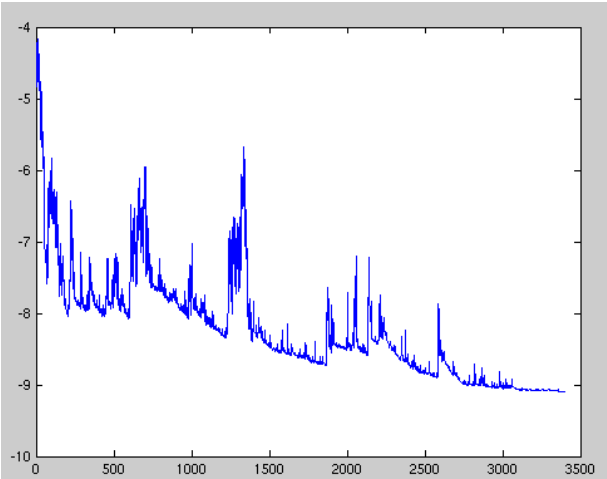

圖. 2.1: SGD fluctuation

2.3 Mini-batch gradient descent

Mini-batch gradient descent(小批量梯度下降) 各取前兩者的優點，將資料集分割成小區塊，每個小區塊大小稱作 batch size，每次跑完 batch size 算迭代 (iteration) 一次，算完一次資料集即完成一次 epoch。舉例: 資料集大小為 1000，若 batch size 為 50，iteration 為 $\frac{datasets}{batch\_size} = \frac{1000}{50} = 20$，當 iteration 跑完 20 次算完成一次 epoch。

這方式可以減少參數更新的方差，並且可以穩定收斂；可利用最先進的深度學習庫所共有的高度優化的矩陣優化，從而由一個小批量計算出梯度非常有效。通常 batch sizes 的範圍介於 50 ~256，會因為應用而有所差異。訓練神經網絡時，通常選擇 Mini-batch gradient descent 算法，而當使用這算法時，通常也使用術語 SGD。

$$\theta = \theta - \eta \cdot \bigtriangledown_\theta J_{(\theta, x^{(i:i+n)}, y^{(i:i+n)})}$$

下面程式碼 [程式.2.3] 為迭代範例，batch size 大小為 50：

程式. 2.3: Mini-batch gradient descent

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

2.4 Challenges

Mini-batch gradient descent 無論如何還是無法確保收斂的很好，存在一些需要解決的挑戰：

1. 選擇適當的學習率是有難度的。如果學習率太小會導致收斂困難或緩慢，學習率太大則會阻礙收斂導致 loss function 來回波動或發生偏離。

2. 學習率清單嘗試在訓練的時候調整學習率，即根據預定義清單或當目標下降於閾值 (threshold) 時降低學習率。但清單和閾值須預先定義，因此無法適應數據集的特徵。

3. 另外相同學習率適用全部參數更新。如果資料稀疏而且外型有很特別的頻率，我們可能不希望將所有特徵更新到相同的程度，而是對很少發生的特徵執行較大的更新。

4. 最小化神經網路常見的高度非凸面誤差方程式 (error function) 的另一關鍵挑戰則是要避免被困在大量次優的局部最小值區域中。認為困難實際上不是由局部最小值引起的，而是由鞍點引起的，即一維向上傾斜而另一維向下傾斜的點。這些鞍點通常被相同誤差的平穩段包圍，這使得 SGD 很難逃脫，因為在所有維度上梯度都接近於零。

Chapter 3

Gradient descent optimization algorithms

在下文中，整理概述深度學習社區廣泛使用的一些算法來應對上述挑戰。我們不會討論在實際中無法計算高維數據集的算法，例如二階法、牛頓法。

3.1 Momentum

SGD 難以在陡峭的往正確的方向，那就是說在一個維度上，曲面的彎曲比另一個維度要陡得多，這在局部最優情況下很常見。在這些情況下，SGD 會在陡峭的地方振盪，而僅沿著底部朝著局部最優方向猶豫前進，如 [圖.3.1(a)] 所示。

Momentun(動量) 是一個幫助加速 SGD 在正確方向和抑制震盪的方法，在 [圖.3.1(b)]。

(a) SGD without momentum　　(b) SGD with momentum

圖. 3.1: SGD momentum

這麼做會增加一個係數 γ 來更新上次的向量到正確向量 (修正偏差)，γ 通常設為 0.9 左右。

$$v_t = \gamma v_{t-1} + \eta \cdot \bigtriangledown_\theta J_{(\theta)}$$

$$\theta = \theta - v_t$$

實際上，使用動量的時候，就像將球推下山坡。球在下坡時滾動時會累積動量，在途中速度會越來越快（如果存在空氣阻力，直到達到極限速度，也就是 $\gamma < 1$) 參數更新也發生了同樣的事情：動量 (momentum) 對於梯度指向相同方向的維度增加，而對於梯度改變方向的維減少動量。結果，我們獲得了更快的收斂並減少了振盪。

3.2 Nesterov accelerated gradient

然而，Momentun 的做法就像，一個球從山上滾下來，盲目地跟隨斜坡。我們希望有一個更聰明的球，這個球有一個去向的概念，以便在山坡再次變高之前知道它會減速。

Nesterov accelerated gradient（NAG）是一種使動量具有這種先見之明的方式。我們知道使用動量 γv_{t-1} 來移動參數 θ。計算 $\theta - \gamma v_{t-1}$ 這樣就給了參數的下一個位置的近似值（完整更新缺少的梯度），這是參數將要存在的大致概念。現在，通過計算與當前參數無關的梯度來有效地看到目前的參數 θ 將會移動到的位置：

$$v_t = \gamma v_{t-1} + \eta \cdot \bigtriangledown_\theta J_{(\theta - \gamma v_{t-1})}$$

$$\theta = \theta - v_t$$

同樣，我們設置動量 γ 約為 0.9。動量首先計算當前梯度（[圖.3.2] 中的藍色小向量），然後在更新的累積梯度（藍色向量）的方向上發生較大的跳躍，而 NAG 首先在先前的累積梯度的方向上進行較大的跳躍（棕色向量），測量梯度，然後進行校正（紅色向量），從而完成 NAG 更新（綠色向量）。這種預期的更新可防止我們過快地進行，並導致響應速度增加，從而顯著提高了 RNN 在許多任務上的性能。有關 NAG 背後另一解釋，請參見此處，而 Ilya Sutskever 在其博士論文中給出了更詳細的概述。

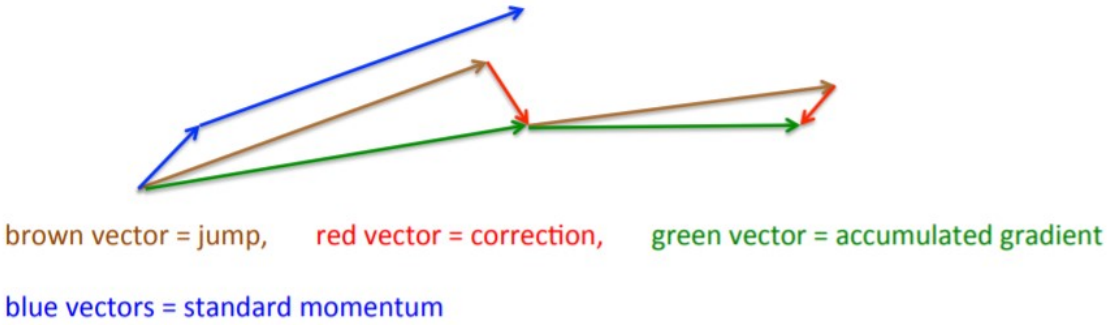

圖. 3.2: NAG

既然能夠使更新適應誤差函數的斜率並依次提高 SGD，我們還希望使更新適應每個單獨的參數，以根據其重要性執行更大或更小的更新。

3.3 Adagrad

Adagrad 是一個梯度優化的算法，它可以做到：學習率適應參數，對於頻繁出現的特徵相關參數執行較小的更新 (較低的學習率)，以及對不經常出現的特徵相關參數進行較大更新（即學習率較高）。Adagrad 可以提高 SGD 的強度，用於訓練大型神經網絡。

先前，在同一次 θ 參數 (更新後就算另一次)，每個 θ 都使用相同的 η(學習率)。Adagrad 則是對每個 θ 參數使用不同的 η，t 代表 time step。先將 Adagrad 的參數更新向量化。用 g_t 表示 time step 的梯度，$g_{t,i}$ 表示目標函數 (參數 θ 在 time step t) 對參數做偏微分計算。

$$g_{t,i} = \bigtriangledown_\theta J_{(\theta_{t,i})}$$

當 SGD 更新每個參數 θ_i，在每個 time step t，因此變成：

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}$$

更新規則，Adagrad 根據先前 θ_i 計算的梯度，對每個參數 θ_i 修改整個學習率 η 在每個 time step$t$：

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}}$$

$G_t \in \mathbb{R}^{d \times d}$ 這是一個對角矩陣每個對角元素 i，i 是關於 θ 梯度平方和取決於 time step$t$，ϵ 是避免分母為 0(ϵ 通常為 1×10^{-8})，如果沒有平方根運算，該算法的性能將大大降低。

G_t 包含了過去梯度平方根，由於全部 θ 參數沿著對角線，通過向量的內積計算 G_t 和 g_t：

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \cdot g_t$$

Adagrad 主要好處之一是，無需手動調整學習率。大多數實現使用預設值 0.01 並將其保留為預設值。Adagrad 主要弱點是會累積分母的平方梯度：由於每項都是正的，累積和會在訓練中不斷增長。反過來，學習率下降，並最終變得無限小，這算法就不再獲得知識。

3.4 Adadelta

Adadelta 是 Adagrad 的延伸，下降其激進的程度，單調的降低學習率。Adadelta 會限制過去累積的梯度，並將其限制在某個特定大小 w，並代替 Adagrad 過去累積的梯度平方。不是低效的存儲 w 先前的平方梯度，而是梯度總和是遞迴定義為所有過去衰減梯度平方平均值。流動平均 $E[g^2]_t$ 在 time step t 然後取決於 (像 Momentum 的 γ) 先前平均和最近梯度：

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma)g_t^2$$

γ 值和 Momentum 的相似，約為 0.9，現在根據參數更新向量 $\triangle\theta_t$ 來重寫 SGD：

$$\triangle\theta_t = -\eta \cdot g_{t,i}$$

$$\theta_{t+1} = \theta_t + \triangle\theta_t$$

Adagrad 的參數更新向量替換成：對角矩陣 G_t 過去梯度平方的衰退平均 $E[g^2]_t$

$$\triangle\theta_t = -\frac{\eta}{\sqrt{G_t+\epsilon}} \cdot g_t$$

$$replace\ G_t\ with\ E[g^2]_t \Rightarrow \triangle\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t+\epsilon}} \cdot g_t$$

由於分母只是梯度的均方根 (RMS)，我們可以取代成縮寫：

$$\triangle\theta_t = -\frac{\eta}{RMS[g]_t} \cdot g_t$$

這個更新單位和 SGD、Momentum 以及 Adagrad 的單位不符合，因此更新需有相同的參數。為了實現這一點，首先定義另一個指數衰減平均值，這次不是梯度平方更新而是參數平方更新：

$$E[\triangle\theta^2]_t = \gamma E[\triangle\theta^2]_{t-1} + (1-\gamma) \triangle \theta_t^2$$

RMS 參數更新:

$$RMS[\triangle\theta]_t = \sqrt{E[\triangle\theta^2]_t+\epsilon}$$

$RMS[\triangle\theta]_t$ 是未知的，更新參數的 RMS 取近似直到上個 time step。用 $RMS[\triangle\theta]_{t-1}$ 取代學習率 η，最後產生新的規則：

$$\triangle\theta_t = -\frac{RMS[\triangle\theta]_{t-1}}{RMS[g]_t} g_t$$

$$\theta_{t+1} = \theta_t + \triangle\theta_t$$

使用 Adadelta，甚至不需要設定預設學習率，因為它已從更新規則淘汰。

3.5 RMSprop

RMSprop 是 Geoffrey Hinton 在他的課程中提出的未公開自適應學習率的方法。

RMSprop 和 Adadelta 都是為了解決 Adagrad 的學習率急劇下降的問題個別獨立開發出來的解決方式。RMSprop 實際上與 Adadelta 得出的第一個更新向量相同：

$$E[g^2]_t = 0.9E[g^2]_t + 0.1g_t^2$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t+\epsilon}} g_t$$

RMSprop 也將學習率除以梯度平方的指數衰減平均值。Hinton 建議 γ 設為 0.9，好的預設學習率 η 數值為 0.001。

3.6 Adam

Adaptive Moment Estimation 自適應矩評估 (Adam) 是另一種計算每個評估學習率的方法。出了儲存過去梯度平方的指數衰減平均值 v_t，就像 Adadelta 和 RMSprop 一樣，Adam 還保留過去梯度的指數衰減平均值 m_t，類似動量 (Momentum)。如果 Momentum 被視為順著斜坡下滑的球，而 Adam 則是像一個帶有摩擦的沉重的球，因此更適合待在 error face 平坦的最小值區域。計算過去梯度平方的衰減平均值 m_t 和 v_t 分別如下：

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

m_t 和 v_t 分別是第一階矩平均估計值和第二階矩無中心方差估計值，因此是方法的名稱。像 m_t 和 v_t 被初始化為向量 o，Adam 的作者觀察到它們偏向零，特別是在初始 time step，尤其是在衰減率較小的時候 (也就是說 β_1 和 β_2 趨近於 1) 藉由計算校正偏差第一矩和第二矩抵消偏差：

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

使用他們去更新參數，就像 Adadelta 和 RMSprop 中所看到的那樣，這將產生 Adam 更新規則：

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

β_1 預設值建議為 0.9，β_2 預設值建議為 0.999，ϵ 預設值建議為 10^{-8}。根據經驗證明 Adam 表現良好，並且與其他自適應學習算法相比具有優勢。

3.7 AdaMax

在 Adam 更新規則中的 v_t 係數是與梯度成反比地縮放過去梯度的範數 (通過 v_{t-1} 項) 和當前梯度 $|g_t|^2$：

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) |g_t|^2$$

我們轉換這個更新到 ℓ_p。注意 β_2 參數化為 β_2^p：

$$v_t = \beta_2^p v_{t-1} + (1 - \beta_2^p) |g_t|^p$$

大規範 p 值使數值上變得不穩定，這就是為什麼 ℓ_1 和 ℓ_2 規範在實踐中是最常見的。然而，ℓ_∞ 通常也表現出穩定的行為。作者 (Kingma and Ba, 2015) 提出了 AdaMax 並證明了和 ℓ_∞ 收斂到更穩定的值。為了避免與 Adam 混用，所以使用 u_t 來表示無窮範數約束 v_t：

$$u_t = \beta_2^\infty v_{t-1} + (1 - \beta_2^\infty) |g_t|^\infty$$

$$= \max(\beta_2 \cdot v_{t-1}, |g_t|)$$

替換為 Adam 更新公式 $\sqrt{\hat{v}_t} + \epsilon$ 和 u_t 得出 AdaMax 更新規則：

$$\theta_{t+1} = \theta_t - \frac{\eta}{u_t} \hat{m}_t$$

注意 u_t 依靠最大運算，不建議 Adam 中的 m_t 和 v_t 偏向零，這就是為什麼不需要針對 u_t 計算偏差。好的預設值 $\eta = 0.002$，$\beta_1 = 0.9$ 和 $\beta_2 = 0.999$。

3.8 Nadam

Adam 可以看作是 RMSprop 和的組合：RMSprop 貢獻了過去梯度平方的指數衰減平均值 v_t，而 Momentum 則代表過去梯度指數的衰減平均值。還看到 Nesterov accelerated gradient (NAG) 優於 Vanilla momentum。

Nadam (Nesterov-accelerated Adaptive Moment Estimation，Nesterov 加速的自適應矩估計)，因此結合了 Adam 和 NAG。為了將 NAG 納入 Adam，需要修改動量項 m_t。使用先前符號回顧動量更新規則：

$$g_t = \nabla_{\theta_t} J(\theta_t)$$

$$m_t = \gamma m_{t-1} + \eta g_t$$

$$\theta_{t+1} = \theta_t - m_t$$

J 是目標函數，γ 是動量衰減項，η 是 step size(學習率)，上面的第三個方程式擴展為：

$$\theta_{t+1} = \theta_t - (\gamma m_{t-1} + \eta g_t)$$

再次證明了動量涉及在前一個動量向量的方向上往前一步和在當前梯度的方向上邁出一步。NAG 然後允許計算梯度之前透過更新動量步長參數使梯度方向上執行更精確的步長。因此，我們只需要修改梯度 g_t 到達 NAG：

$$g_t = \nabla_{\theta_t} J(\theta_t - \gamma m_{t-1})$$

$$m_t = \gamma m_{t-1} + \eta g_t$$

$$\theta_{t+1} = \theta_t - m_t$$

Dozat 建議以以下方式修改 NAG：一次用於更新梯度 g_t 第二次更新參數 θ_{t+1}，而不是應用 momentum step 兩次，現在直接應用前瞻動量向量來更新當前參數：

$$g_t = \nabla_{\theta_t} J(\theta_t)$$

$$m_t = \gamma m_{t-1} + \eta g_t$$

$$\theta_{t+1} = \theta_t - (\gamma m_t + \eta g_t)$$

為了將 Nesterov 動量添加到 Adam，可以類似地用當前動量向量替換以前的動量向量。回想一下 Adam 更新規則如下：

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

用定義拓展第二個方程式：

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} (\frac{\beta_1 m_{t-1}}{1 - \beta_1^t} + \frac{(1 - \beta_1) g_t}{1 - \beta_1^t})$$

注意 $\frac{\beta_1 m_{t-1}}{1-\beta_1^t}$ 只是前一個的 time step 的動量向量的偏差來校正評估。因此，可以將其替換為 $\hat{m}_{t-1}$：

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} (\beta_1 \hat{m}_{t-1} + \frac{(1 - \beta_1) g_t}{1 - \beta_1^t})$$

為簡化，因為無論如何將在下一步中替換分母，所以忽略了分母 $1 - \beta_1^t$。該方程式再次看起來和上面擴展的動量更新規則非常相似。可以像以前一樣添加 Nesterov 動量，方法是用當前動量向量偏差校正後的評估值替換前一時間步長的動量向量偏差校正後的評估值，這為我們提供了 Nadam 更新規則：

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} (\beta_1 \hat{m}_t + \frac{(1 - \beta_1) g_t}{1 - \beta_1^t})$$

3.9 AMSGrad

Reddi 等（2018）。正式化了這個問題，並指出了泛化行為不佳的原因：將過去梯度平方的指數移動平均值作為自適應學習率方法。雖然引入指數平均值的動機很充分：應防止學習率隨著訓練的進行而變得無限小；但這也是 Adagrad 算法的關鍵缺陷。在其他情況下，短期記憶的梯度成為障礙。

在 Adam 收斂到次優解的環境中，已經觀察到一些小型批次提供了較大且信息豐富的梯度，但是由於這些小型批次很少出現，因此指數平均會減小其影響，從而導致收斂性較差。作者 (資料來源的作者) 提供了一個簡單的凸型優化問題的例子，其中 Adam 可以觀察到相同的行為。

為了解決此問題，作者提出了一種新算法 AMSGrad，該算法使用了過去梯度平方的最大值 v_t 而不是指數平均值來更新參數。v_t 的定義與先前的 Adam 相同：

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

而不是直接使用 v_t(或其偏差更正的版本 $\hat{v}_t$)，如果現在使用以前值的大於現在的值：

$$\hat{v}_t = \max(\hat{v}_{t-1}, v_t)$$

這方式 AMSGrad 的不會增加步長 (step size)，從而避免了 Adam 遇到的問題。為了簡化，AMSGrad 去除了 Adam 的去偏差 (debias) 步驟。可以看到完整的 AMSGrad 更新，沒有經過偏差校正的估計：

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

$$\hat{v}_t = \max(\hat{v}_{t-1}, v_t)$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} m_t$$

在小型數據集和 CIFAR-10 上，與 Adam 相比，性能有所提高。但是，其他實驗顯示其性能與 Adam 相似或更差。在實際運用，AMSGrad 是否能勝過 Adam，還有待觀察。有關深度學習優化的最新進展的更多信息，請參閱此blog 文章。

3.10 Gradient noise

增加 noise 跟隨高斯分布 $N(0, \sigma_t^2)$ 對每個梯度更新：

$$g_{t,i} = g_{t,i} + N(0, \sigma_t^2)$$

根據排定時間對差異計算：

$$\sigma_t^2 = \frac{\eta}{(1 + t)^\gamma}$$

添加這種 noise 對不良初始化的網絡可使其更強化，並有助於訓練特別深且復雜的網絡。他們懷疑增加的噪聲使模型有更多的機會逃脫並找到新的局部極小值，這對於更深的模型而言更常見。

3.11 其他最近的優化器

在 AMSGrad 之後，已經提出了許多其他優化器。其中包括 AdamW ，可以修復 Adam 的權重下降；QHAdam，將標準 SGD 步長與動量 SGD 步長進行平均；和 AggMo，它結合了多個動量項 γ 和別的。有關最近的梯度下降算法的概述，請參閱此blog 文章。